A SMALL HARDWARE IMPLEMENTATION OF THE SUBBYTE FUNCTION OF RIJNDAEL

1. Field of the Invention

The present invention relates to the field of data encryption. The invention relates particularly to an apparatus and method for a small hardware implementation of the SubByte function found in the Advanced Encryption Standard (AES) algorithm or Rijndael Block Cipher, hereinafter AES/Rijndael. The accommodating is redesigned to work with both inverse and normal processing.

2. Discussion of the Related Art

The current state of the art provides for hardware implementations where the inverse cipher can only partially re-use the circuitry that implements the cipher. For high-speed networking processors and Smart Card applications a very small (gate size) and high data-rate (accommodating an Optical Carrier Rate of OC-192 and beyond 9953.28 Mbps and a payload of 9.6 Gbps) are desirable.

The AES/Rijndael is an iterataed block cipher and is described in a proposal written by Joan Daemen and Vincent Rijmen and published in March 9, 1999. The National Institute of Standards and Technology (NIST) has approved the AES/Rijndael as a cryptographic algorithm and published the AES/Rijndael in November 26, 2001 (Publication 197 also known as Federal Information Processing Standard 197 or "FIPS 197") which is hereby incorporated by reference as if fully set forth herein). In accordance with many private key encryption/decryption algorithms, including AES/Rijndael, encryption/decryption is performed in multiple stages, commonly known as iterations or rounds. Such algorithms lend themselves to a data processing pipeline or pipelines architecture. In each round, the AES/Rijndael uses the affine transformation and its inverse along with other transformations to decrypt (decipher) and encrypt (encipher) information.

Encryption converts data to an unintelligible form called cipher text; decrypting the ciphertext converts the data back into its original form, called plaintext.

The input and output for the AES/Rijndael algorithm each consist of sequences of 128 bits (each having a value of 0 or 1). These sequences are commonly be referred to as blocks and the number of bits they contain are referred to as their length ("FIPS 197", NIST, p. 7). The basic unit for processing in the AES/Rijndael algorithm is a byte, a sequence of eight bits treated as a single entity with most significant bit (MSB) on the left. Internally, the AES/Rijndael algorithm's operations are performed on a two dimensional array of bytes called the State. The State consists of four rows of bytes, each containing Nb bytes, where Nb is the block length divided by 32 ("FIPS 197", NIST, p. 9).

At the start of the Cipher and Inverse Cipher (encryption and decryption), the input - the array of bytes

$in_0, in_1, \ldots in_{15}$

is copied into the State array as illustrated in FIG 1. The Cipher or Inverse Cipher operations are then conducted on each byte in this State array, after which its final values are copied to the output -- the array of bytes

$out_0, out_1, \ldots out_{15}$.

The addition of two elements in a finite field is achieved by "adding" the coefficients for the corresponding powers in the polynomials for the two elements. The addition is performed with the boolean exclusive XOR operation ("FIPS 197",NIST,p 10). The binary notation for adding two bytes is:

$$\{01010111\} \oplus \{10000011\} = \{11010100\} \quad (1.0)$$

In the polynomial representation, multiplication in $GF(2^8)$ corresponds with the multiplication of polynomials modulo an irreducible polynomial of degree 8. A polynomial

is irreducible if its only divisors are one and itself. For the AES/Rijndael algorithm, this irreducible polynomial is ("FIPS 197", NIST, p.10):

$$m\langle x\rangle = x^8 + x^4 + x^3 + x + 1 \quad (1.1)$$

A diagonal matrix with each diagonal element equal to 1 is called an ***identity matrix***. The n x n identity matrix is denoted $\mathbf{I_n}$:

$$I_n = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.2)$$

If **A** and **B** and n x n matrices, we call each an ***inverse*** of the other if:

$$AB = BA = I_n \quad (1.3)$$

A transformation consisting of multiplication by a matrix followed by the addition of a vector is called an *Affine Transformation*.

The SubByte() function of AES/Rijndael is a non-linear byte substitution that operates independently on each byte of the State using a substitution table (S-box). This S-box, which is invertible, is constructed by composing two transformations:

1. Take the multiplicative inverse in the finite field GF(2^8), described earlier; the element {00} is mapped to itself.
2. Apply the following affine transformation (over GF(2)):

$$b_i' = b_{(i) \bmod 8} \oplus b_{(i+4) \bmod 8} \oplus b_{(i+5) \bmod 8} \oplus b_{(i+6) \bmod 8} \oplus b_{(i+7) \bmod 8} \oplus c_i \quad (1.4)$$

In matrix form, the affine transformation element of the S-box can be expressed as ("FIPS 197",NIST,p16):

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \quad (1.5)$$

If this were implemented as the lookup table as suggested by the AES/Rijndael proposal, a 256 entry ROM or multiplexor would be required. To implement the AES/Rijndael algorithm, 12 instantiations of this table would be required. The inverse of this matrix can be found as:

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad (1.6)$$

If this was implemented as the lookup table suggested by the AES/Rijndael proposal, a 128-entry, 16-bit word ROM or multiplexor would be required. To implement the AES/Rijndael algorithm, 12 instantiations of this table would be required.

Thus there is a need for a system and a method of sharing almost all the circuitry for the affine transformation in order to reduce gate count. To achieve a high data-rate and small gate size the design must be architected so that the maximum path is not significantly longer and the gate size is so small that the design can be replicated to promote parallel processing without greatly increasing the die size. Increasing die size adds more expense

and power consumption, making the product less marketable. The present invention is an apparatus and a method for decreasing the gate size and at the expense of slightly increasing the maximum path delay. This makes the circuit smaller and thus more attractive for high data-rate designs.

Each occurrence in the AES/Rijndael of the pair of affine transform and inverse affine transform is reduced by the present invention to one transform, the Affine-All transform. In a preferred embodiment, a circuit performs both normal and inverse affine transformations with very little duplicate logic. In this preferred embodiment, by implementing the Affine-All transform with a Multiplicative Inverse ROM, the logic is greatly reduced and the maximum path delay is reduced compared to a multiplexor implementation while only being slightly greater than for a ROM implementation

Thus, the preferred embodiment of the present invention employs a read-only memory (ROM) for the multiplicative inverse and a reduced combinational logic implementation for the affine transformation. This implementation is very low in gate count with a very comparable maximum delay path.

FIG. 1 illustrates state array input and output ("FIPS 197", nist, p.9)

FIG. 2 illustrates comparison of prior art ROM and lookup table (multiplexor) implementation of the subbyte function with Affine-All implementation of the present invention.

FIG. 3 illustrates the ROM or lookup table used with the Affine-All transformation of the present invention.

FIG. 4 illustrates the netlist of the Affine-All combinational logic.

The present invention is based, in part, on the fact that beginning at the last row each row of matrix equations (1.5) and (1.6) is shifted left by one bit from the previous row. In the present invention, the first row of each matrix is termed the "load pattern". So the "load

pattern" for the affine transform matrix is {10001111} and the "load pattern" for the inverse affine transform is {00100101}. Note that the number of 0's in each "load pattern" is an odd number and is an important characteristic in being able to merge the two transformations into one circuit in the system and method of the present invention.

If both affine transformations are implemented as suggested by Daemen and Rijmen ("FIPS 197") using exclusive OR gates the circuit equations look as follows:

Affine Transform Equations

$$
\begin{aligned}
b'_0 &= 5(b_0 \rho b_4 \rho b_5 \rho b_6 \rho b_7) \\
b'_1 &= 5(b_0 \rho b_1 \rho b_5 \rho b_6 \rho b_7) \\
b'_2 &= (b_0 \rho b_1 \rho b_2 \rho b_6 \rho b_7) \\
b'_3 &= (b_0 \rho b_1 \rho b_2 \rho b_3 \rho b_7) \\
b'_4 &= (b_0 \rho b_1 \rho b_2 \rho b_3 \rho b_4) \\
b'_5 &= 5(b_1 \rho b_2 \rho b_3 \rho b_4 \rho b_5) \\
b'_6 &= 5(b_2 \rho b_3 \rho b_4 \rho b_5 \rho b_6) \\
b'_7 &= (b_3 \rho b_4 \rho b_5 \rho b_6 \rho b_7)
\end{aligned}
\qquad \textbf{(1.7)}
$$

Notice that each equation has an odd number of terms and the same number of terms: five. The addition of the vector determines the negation of some equations. So the number of terms in each equation is determined by the "load pattern". The number of negations is determined by the addition of the vector which is termed the "load vector".

Inverse Affine Transform Equations

$$
\begin{aligned}
b'_0 &= 5(b_2 \rho b_5 \rho b_7) \\
b'_1 &= (b_0 \rho b_3 \rho b_6) \\
b'_2 &= 5(b_1 \rho b_4 \rho b_7) \\
b'_3 &= (b_0 \rho b_2 \rho b_5) \\
b'_4 &= (b_1 \rho b_3 \rho b_6) \\
b'_5 &= (b_2 \rho b_4 \rho b_7) \\
b'_6 &= (b_0 \rho b_3 \rho b_5) \\
b'_7 &= (b_1 \rho b_4 \rho b_6)
\end{aligned}
\qquad \textbf{(1.8)}
$$

Each equation has an odd number of terms and the same number of terms: three. The addition of the vector determines the negation of some equations. So the number of terms in

each equation is determined by the "load pattern". The number of negations is determined by the addition of the vector.

This addition vector can now be used as a "load vector" as well. Looking at the two sets of equations it appears that there is no common logic to be merged. If the equations are rewritten with the "load pattern" included and use the addition of the vector to determine the negations, a common circuit is revealed. The properties of the exclusive OR are used to accomplish this and these properties are:

$$A \oplus B \oplus C = C \oplus B \oplus A \quad (1.9)$$

$$A \oplus 0 = A \quad (2.0)$$

$$A \oplus 1 = \neg A \quad (2.1)$$

$$A \oplus A = 0 \quad (2.2)$$

In a preferred embodiment, the circuit implementing both the affine and inverse affine transforms comprises a Multiplicative Inverse ROM and the logic that represents both transforms is as follows with **p** as the "load pattern" and **v** as the "load vector". For example, here is what equation seven of the affine matrix becomes:

$$b'_7=[(b_0 \cong p_1)\rho(b_1 \cong p_2)\rho(b_2 \cong p_3)\rho(b_3 \cong p_4)\rho(b_4 \cong p_5)\rho(b_5 \cong p_6)\rho(b_6 \cong p_7)\rho(b_7 \cong p_0)]\rho v_7 \quad (2.3)$$

The number of instantiations has been cut in half. Because of the 0's produced by the ANDing of **p** and **b**, the equation works for both affine and inverse affine transformations. Because **b** XOR'ed with a 1 is always the inverse of **b**, using v_7 each time negates the equation where appropriate.

Comparisons:

Using the design suggested by the AES/Rijndael proposal (FIPS 197) implemented in two ways:

(1) a 128-entry, 16-bit word ROM, and

(2) a 128-entry, 16-bit word lookup table implemented as a multiplexor,

the ROM, Multiplexor and the Affine-All circuit embodiment of the present invention were synthesized and timed using maximum path analysis. FIG. 2 compares results where sizes in gates are given as well as sizes in microns for comparison with the ROM implementation. Net area is not considered because wire load models differ with technologies.

A preferred embodiment of the ROM or Lookup table contains the values shown in FIG. 3, in hexadecimal format.

The net list of the Affine-All combinational logic of a preferred embodiment is shown in FIG. 4. The code for an implementation is included as Appendix A.

The present invention is applicable to all systems and devices capable of secure communications, comprising security networking processors, secure keyboard devices, magnetic card reader devices, smart card reader devices, and wireless 802.11 devices.

The above describe embodiments are only typical examples, and their modifications and variations are apparent to those skilled in the art. Various modifications to the above-described embodiments can be made without departing from the scope of the invention as embodied in the accompanying claims.

APPENDIX A

The RTL to implement the affine all circuit is shown below:

```
'timescale 10ns/10ns
module aes_affine_all
(
byteOut, // output byte
byteIn, // input byte
enCrypt // 1 = encrypt 0 = decrypt
);
// ----------
// ports
// ----------
```

```
input enCrypt;
input [7:0] byteIn;
output [7:0] byteOut;

// Logic reduction
wire [4:0] byteOut_int;
wire [0:7] y_inv,y,y_int;
wand byteOut_7_0,byteOut_7_1,byteOut_7_2,byteOut_7_3,byteOut_7_4,byteOut_7_5,
byteOut_7_6,byteOut_7_7;
wand byteOut_4_0,byteOut_4_1,byteOut_4_2,byteOut_4_3,byteOut_4_4,byteOut_4_5,
byteOut_4_6,byteOut_4_7;
wand
byteOut_int_4_0,byteOut_int_4_1,byteOut_int_4_2,byteOut_int_4_3,byteOut_int_4_4,
byteOut_int_4_5, byteOut_int_4_6, byteOut_int_4_7;
wand
byteOut_int_3_0,byteOut_int_3_1,byteOut_int_3_2,byteOut_int_3_3,byteOut_int_3_4,
byteOut_int_3_5,byteOut_int_3_6, byteOut_int_3_7;
wand byteOut_3_0,byteOut_3_1,byteOut_3_2,byteOut_3_3,byteOut_3_4,byteOut_3_5,
byteOut_3_6,byteOut_3_7;
wand
byteOut_int_2_0,byteOut_int_2_1,byteOut_int_2_2,byteOut_int_2_3,byteOut_int_2_4,
byteOut_int_2_5,byteOut_int_2_6, byteOut_int_2_7;
wand
byteOut_int_1_0,byteOut_int_1_1,byteOut_int_1_2,byteOut_int_1_3,byteOut_int_1_4,
byteOut_int_1_5,byteOut_int_1_6, byteOut_int_1_7;
wand
byteOut_int_0_0,byteOut_int_0_1,byteOut_int_0_2,byteOut_int_0_3,byteOut_int_0_4,
byteOut_int_0_5,byteOut_int_0_6, byteOut_int_0_7;
assign y_inv = 8'b00100101;
assign y = 8'b10001111;
assign y_int = (enCrypt) ? y : y_inv;
assign byteOut_7_0 = byteIn [0];
assign byteOut_7_0 = y_int[1];
assign byteOut_7_1 = byteIn [1];
assign byteOut_7_1 = y_int[2];
assign byteOut_7_2 = byteIn [2];
assign byteOut_7_2 = y_int[3];
assign byteOut_7_3 = byteIn [3];
assign byteOut_7_3 = y_int[4];
assign byteOut_7_4 = byteIn [4];
assign byteOut_7_4 = y_int[5];
assign byteOut_7_5 = byteIn [5];
assign byteOut_7_5 = y_int[6];
assign byteOut_7_6 = byteIn [6];
assign byteOut_7_6 = y_int[7];
assign byteOut_7_7 = byteIn [7];
assign byteOut_7_7 = y_int[0];
assign byteOut [7] = byteOut_7_0^ byteOut_7_1^ byteOut_7_2^ byteOut_7_3^
byteOut_7_4^ byteOut_7_5^ byteOut_7_6^ byteOut_7_7;
assign byteOut_int_4_0 = byteIn [0];
assign byteOut_int_4_0 = y_int[2];
assign byteOut_int_4_1 = byteIn [1];
```

```
assign byteOut_int_4_1 = y_int[3];
assign byteOut_int_4_2 = byteIn [2];
assign byteOut_int_4_2 = y_int[4];
assign byteOut_int_4_3 = byteIn [3];
assign byteOut_int_4_3 = y_int[5];
assign byteOut_int_4_4 = byteIn [4];
assign byteOut_int_4_4 = y_int[6];
assign byteOut_int_4_5 = byteIn [5];
assign byteOut_int_4_5 = y_int[7];
assign byteOut_int_4_6 = byteIn [6];
assign byteOut_int_4_6 = y_int[0];
assign byteOut_int_4_7 = byteIn [7];
assign byteOut_int_4_7 = y_int[1];
assign byteOut_int [4] = byteOut_int_4_0^ byteOut_int_4_1^ byteOut_int_4_2^
byteOut_int_4_3^ byteOut_int_4_4^
byteOut_int_4_5^ byteOut_int_4_6^ byteOut_int_4_7;
assign byteOut_int_3_0 = byteIn [0];
assign byteOut_int_3_0 = y_int[3];
assign byteOut_int_3_1 = byteIn [1];
assign byteOut_int_3_1 = y_int[4];
assign byteOut_int_3_2 = byteIn [2];
assign byteOut_int_3_2 = y_int[5];
assign byteOut_int_3_3 = byteIn [3];
assign byteOut_int_3_3 = y_int[6];
assign byteOut_int_3_4 = byteIn [4];
assign byteOut_int_3_4 = y_int[7];
assign byteOut_int_3_5 = byteIn [5];
assign byteOut_int_3_5 = y_int[0];
assign byteOut_int_3_6 = byteIn [6];
assign byteOut_int_3_6 = y_int[1];
assign byteOut_int_3_7 = byteIn [7];
assign byteOut_int_3_7 = y_int[2];
assign byteOut_int [3] =byteOut_int_3_0^ byteOut_int_3_1^ byteOut_int_3_2^
byteOut_int_3_3^ byteOut_int_3_4^
byteOut_int_3_5^ byteOut_int_3_6^ byteOut_int_3_7;
assign byteOut_4_0 = byteIn [0];
assign byteOut_4_0 = y_int[4];
assign byteOut_4_1 = byteIn [1];
assign byteOut_4_1 = y_int[5];
assign byteOut_4_2 = byteIn [2];
assign byteOut_4_2 = y_int[6];
assign byteOut_4_3 = byteIn [3];
assign byteOut_4_3 = y_int[7];
assign byteOut_4_4 = byteIn [4];
assign byteOut_4_4 = y_int[0];
assign byteOut_4_5 = byteIn [5];
assign byteOut_4_5 = y_int[1];
assign byteOut_4_6 = byteIn [6];
assign byteOut_4_6 = y_int[2];
assign byteOut_4_7 = byteIn [7];
assign byteOut_4_7 = y_int[3];
```

```
assign byteOut [4] =byteOut_4_0^ byteOut_4_1^ byteOut_4_2^
byteOut_4_3^ byteOut_4_4^
byteOut_4_5^ byteOut_4_6^ byteOut_4_7;
assign byteOut_3_0 = byteIn [0];
assign byteOut_3_0 = y_int[5];
assign byteOut_3_1 = byteIn [1];
assign byteOut_3_1 = y_int[6];
assign byteOut_3_2 = byteIn [2];
assign byteOut_3_2 = y_int[7];
assign byteOut_3_3 = byteIn [3];
assign byteOut_3_3 = y_int[0];
assign byteOut_3_4 = byteIn [4];
assign byteOut_3_4 = y_int[1];
assign byteOut_3_5 = byteIn [5];
assign byteOut_3_5 = y_int[2];
assign byteOut_3_6 = byteIn [6];
assign byteOut_3_6 = y_int[3];
assign byteOut_3_7 = byteIn [7];
assign byteOut_3_7 = y_int[4];
assign byteOut [3] = byteOut_3_0^ byteOut_3_1^ byteOut_3_2^
byteOut_3_3^ byteOut_3_4^
byteOut_3_5^ byteOut_3_6^ byteOut_3_7;
assign byteOut_int_2_0 = byteIn [0];
assign byteOut_int_2_0 = y_int[6];
assign byteOut_int_2_1 = byteIn [1];
assign byteOut_int_2_1 = y_int[7];
assign byteOut_int_2_2 = byteIn [2];
assign byteOut_int_2_2 = y_int[0];
assign byteOut_int_2_3 = byteIn [3];
assign byteOut_int_2_3 = y_int[1];
assign byteOut_int_2_4 = byteIn [4];
assign byteOut_int_2_4 = y_int[2];
assign byteOut_int_2_5 = byteIn [5];
assign byteOut_int_2_5 = y_int[3];
assign byteOut_int_2_6 = byteIn [6];
assign byteOut_int_2_6 = y_int[4];
assign byteOut_int_2_7 = byteIn [7];
assign byteOut_int_2_7 = y_int[5];
assign byteOut_int [2] =(~byteOut_int_2_0 & byteOut_int_2_1| ~byteOut_int_2_1 &
byteOut_int_2_0)^
(~byteOut_int_2_2 & byteOut_int_2_3 | ~byteOut_int_2_3 & byteOut_int_2_2)^
(~byteOut_int_2_4 & byteOut_int_2_5 | ~byteOut_int_2_5 & byteOut_int_2_4)^
(~byteOut_int_2_6&byteOut_int_2_7 | ~byteOut_int_2_7&byteOut_int_2_6);
assign byteOut_int_1_0 = byteIn [0];
assign byteOut_int_1_0 = y_int[7];
assign byteOut_int_1_1 = byteIn [1];
assign byteOut_int_1_1 = y_int[0];
assign byteOut_int_1_2 = byteIn [2];
assign byteOut_int_1_2 = y_int[1];
assign byteOut_int_1_3 = byteIn [3];
assign byteOut_int_1_3 = y_int[2];
```

```
assign byteOut_int_1_4 = byteIn [4];
assign byteOut_int_1_4 = y_int[3];
assign byteOut_int_1_5 = byteIn [5];
assign byteOut_int_1_5 = y_int[4];
assign byteOut_int_1_6 = byteIn [6];
assign byteOut_int_1_6 = y_int[5];
assign byteOut_int_1_7 = byteIn [7];
assign byteOut_int_1_7 = y_int[6];
assign byteOut_int [1] =byteOut_int_1_0^ byteOut_int_1_1^ byteOut_int_1_2^
byteOut_int_1_3^ byteOut_int_1_4^
byteOut_int_1_5^ byteOut_int_1_6^ byteOut_int_1_7;
assign byteOut_int_0_0 = byteIn [0];
assign byteOut_int_0_0 = y_int[0];
assign byteOut_int_0_1 = byteIn [1];
assign byteOut_int_0_1 = y_int[1];
assign byteOut_int_0_2 = byteIn [2];
assign byteOut_int_0_2 = y_int[2];
assign byteOut_int_0_3 = byteIn [3];
assign byteOut_int_0_3 = y_int[3];
assign byteOut_int_0_4 = byteIn [4];
assign byteOut_int_0_4 = y_int[4];
assign byteOut_int_0_5 = byteIn [5];
assign byteOut_int_0_5 = y_int[5];
assign byteOut_int_0_6 = byteIn [6];
assign byteOut_int_0_6 = y_int[6];
assign byteOut_int_0_7 = byteIn [7];
assign byteOut_int_0_7 = y_int[7];
assign byteOut_int [0] =byteOut_int_0_0^ byteOut_int_0_1 ^ byteOut_int_0_2^
byteOut_int_0_3^ byteOut_int_0_4^
byteOut_int_0_5^ byteOut_int_0_6^ byteOut_int_0_7;
assign byteOut [6] = (enCrypt) ? ~byteOut_int[4]: byteOut_int[4];
assign byteOut [5] = (enCrypt) ? ~byteOut_int[3]: byteOut_int[3];
assign byteOut [2] = (enCrypt) ? byteOut_int [2] : ~byteOut_int [2];
assign byteOut [1] = (enCrypt) ? ~byteOut_int[1]: byteOut_int[1];
assign byteOut [0] = ~byteOut_int [0];
endmodule
```